Databricks Spark Reference Applications

databricks

Published with GitBook

Table of Contents

1. Introduction
2. Log Analysis with Spark
    i. Section 1: Introduction to Apache Spark
        i. First Log Analyzer in Spark
        ii. Spark SQL
        iii. Spark Streaming
            i. Windowed Calculations: window()
            ii. Cumulative Calculations: updateStateByKey()
            iii. Reusing Code from Batching: transform()
    ii. Section 2: Importing Data
        i. Batch Import
            i. Importing from Files
                i. S3
                ii. HDFS
            ii. Importing from Databases
        ii. Streaming Import
            i. Built In Methods for Streaming Import
            ii. Kafka
    iii. Section 3: Exporting Data
        i. Small Datasets
        ii. Large Datasets
            i. Save the RDD to Files
            ii. Save the RDD to a Database
    iv. Section 4: Log Analyzer Application
3. Twitter Streaming Language Classifier
    i. Collect a Dataset of Tweets
    ii. Examine the Tweets and Train a Model
        i. Examine with Spark SQL
        ii. Train with Spark MLLib
        iii. Run Examine And Train
    iii. Apply the Model in Real-time
4. Weather TimeSeries Data Application with Cassandra
    i. Overview
    ii. Running the Example

Databricks Reference Apps

At Databricks, we are developing a set of reference applications that demonstrate how to use Apache Spark. This book/repo contains the reference applications.

- View the code in the Github Repo here: https://github.com/databricks/reference-apps
- Read the documentation here: http://databricks.gitbooks.io/databricks-spark-reference-applications/
- Submit feedback or issues here: https://github.com/databricks/reference-apps/issues

The reference applications will appeal to those who want to learn Spark and learn better by example. Browse the applications, see what features of the reference applications are similar to the features you want to build, and refashion the code samples for your needs. Additionally, this is meant to be a practical guide for using Spark in your systems, so the applications mention other technologies that are compatible with Spark - such as what file systems to use for storing your massive data sets.

- Log Analysis Application - The log analysis reference application contains a series of tutorials for learning Spark by example as well as a final application that can be used to monitor Apache access logs. The examples use Spark in batch mode, cover Spark SQL, as well as Spark Streaming.

- Twitter Streaming Language Classifier - This application demonstrates how to fetch and train a language classifier for Tweets using Spark MLLib. Then Spark Streaming is used to call the trained classifier and filter out live tweets that match a specified cluster. For directions on how to build and run this app - see twitter_classifier/scala/README.md.

- Weather TimeSeries Data Application with Cassandra - This reference application works with Weather Data which is taken for a given weather station at a given point in time. The app demonstrates several strategies for leveraging Spark Streaming integrated with Apache Cassandra and Apache Kafka for fast, fault-tolerant, streaming computations with time series data.

These reference apps are covered by license terms covered here.

Log Analysis with Spark

This project demonstrates how easy it is to do log analysis with Apache Spark.

Log analysis is an ideal use case for Spark. It's a very large, common data source and contains a rich set of information. Spark allows you to store your logs in files to disk cheaply, while still providing a quick and simple way to process them. We hope this project will show you how to use Apache Spark on your organization's production logs and fully harness the power of that data. Log data can be used for monitoring your servers, improving business and customer intelligence, building recommendation systems, preventing fraud, and much more.

How to use this project

This project is broken up into sections with bite-sized examples for demonstrating new Spark functionality for log processing. This makes the examples easy to run and learn as they cover just one new topic at a time. At the end, we assemble some of these examples to form a sample log analysis application.

Section 1: Introduction to Apache Spark

The Apache Spark library is introduced, as well as Spark SQL and Spark Streaming. By the end of this chapter, a reader will know how to call transformations and actions and work with RDDs and DStreams.

Section 2: Importing Data

This section includes examples to illustrate how to get data into Spark and starts covering concepts of distributed computing. The examples are all suitable for datasets that are too large to be processed on one machine.

Section 3: Exporting Data

This section includes examples to illustrate how to get data out of Spark. Again, concepts of a distributed computing environment are reinforced, and the examples are suitable for large datasets.

Section 4: Logs Analyzer Application

This section puts together some of the code in the other chapters to form a sample log analysis application.

More to come...

While that's all for now, there's definitely more to come over time.

Section 1: Introduction to Apache Spark

In this section, we demonstrate how simple it is to analyze web logs using Apache Spark. We'll show how to load a Resilient Distributed Dataset (**RDD**) of access log lines and use Spark tranformations and actions to compute some statistics for web server monitoring. In the process, we'll introduce the Spark SQL and the Spark Streaming libraries.

In this explanation, the code snippets are in Java 8. However, there is also sample code in Java 6, Scala, and Python included in this directory. In those folders are README's for instructions on how to build and run those examples, and the necessary build files with all the required dependencies.

This chapter covers the following topics:

1. First Log Analyzer in Spark - This is a first Spark standalone logs analysis application.
2. Spark SQL - This example does the same thing as the above example, but uses SQL syntax instead of Spark transformations and actions.
3. Spark Streaming - This example covers how to calculate log statistics using the streaming library.

First Logs Analyzer in Spark

Before beginning this section, go through Spark Quick Start and familiarize with the Spark Programming Guide first.

This section requires a dependency on the Spark Core library in the maven file - note update this dependency based on the version of Spark you have installed:

```
<dependency> <!-- Spark -->
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.10</artifactId>
    <version>1.1.0</version>
</dependency>
```

Before we can begin, we need two things:

- **An Apache access log file**: If you have one, it's more interesting to use real data.
    - This is trivial sample one provided at data/apache.access.log.
    - Or download a better example here: http://www.monitorware.com/en/logsamples/apache.php
- **A parser and model for the log file**: See ApacheAccessLog.java.

The example code uses an Apache access log file since that's a well known and common log format. It would be easy to rewrite the parser for a different log format if you have data in another log format.

The following statistics will be computed:

- The average, min, and max content size of responses returned from the server.
- A count of response code's returned.
- All IPAddresses that have accessed this server more than N times.
- The top endpoints requested by count.

Let's first walk through the code first before running the example at LogAnalyzer.java.

The main body of a simple Spark application is below. The first step is to bring up a Spark context. Then the Spark context can load data from a text file as an RDD, which it can then process. Finally, before exiting the function, the Spark context is stopped.

```
public class LogAnalyzer {
  public static void main(String[] args) {
    // Create a Spark Context.
    SparkConf conf = new SparkConf().setAppName("Log Analyzer");
    JavaSparkContext sc = new JavaSparkContext(conf);

    // Load the text file into Spark.
    if (args.length == 0) {
      System.out.println("Must specify an access logs file.");
      System.exit(-1);
    }
    String logFile = args[0];
    JavaRDD<String> logLines = sc.textFile(logFile);

    // TODO: Insert code here for processing logs.

    sc.stop();
  }
}
```

Given an RDD of log lines, use the `map` function to transform each line to an ApacheAccessLog object. The

ApacheAccessLog RDD is cached in memory, since multiple transformations and actions will be called on it.

```
// Convert the text log lines to ApacheAccessLog objects and
// cache them since multiple transformations and actions
// will be called on the data.
JavaRDD<ApacheAccessLog> accessLogs =
    logLines.map(ApacheAccessLog::parseFromLogLine).cache();
```

It's useful to define a sum reducer - this is a function that takes in two integers and returns their sum. This is used all over our example.

```
private static Function2<Long, Long, Long> SUM_REDUCER = (a, b) -> a + b;
```

Next, let's calculate the average, minimum, and maximum content size of the response returned. A `map` transformation extracts the content sizes, and then different actions (`reduce`, `count`, `min`, and `max`) are called to output various stats. Again, call `cache` on the context size RDD to avoid recalculating those values for each action called on it.

```
// Calculate statistics based on the content size.
// Note how the contentSizes are cached as well since multiple actions
//   are called on that RDD.
JavaRDD<Long> contentSizes =
   accessLogs.map(ApacheAccessLog::getContentSize).cache();
System.out.println(String.format("Content Size Avg: %s, Min: %s, Max: %s",
    contentSizes.reduce(SUM_REDUCER) / contentSizes.count(),
    contentSizes.min(Comparator.naturalOrder()),
    contentSizes.max(Comparator.naturalOrder())));
```

To compute the response code counts, we have to work with key-value pairs - by using `mapToPair` and `reduceByKey`. Notice that we call `take(100)` instead of `collect()` to gather the final output of the response code counts. Use extreme caution before calling `collect()` on an RDD since all that data will be sent to a single Spark driver and can cause the driver to run out of memory. Even in this case where there are only a limited number of response codes and it seems safe - if there are malformed lines in the Apache access log or a bug in the parser, there could be many invalid response codes to cause an.

```
// Compute Response Code to Count.
List<Tuple2<Integer, Long>> responseCodeToCount = accessLogs
        .mapToPair(log -> new Tuple2<>(log.getResponseCode(), 1L))
        .reduceByKey(SUM_REDUCER)
        .take(100);
System.out.println(String.format("Response code counts: %s", responseCodeToCount));
```

To compute any IPAddress that has accessed this server more than 10 times, we call the `filter` tranformation and then `map` to retrieve only the IPAddress and discard the count. Again we use `take(100)` to retrieve the values.

```
List<String> ipAddresses =
    accessLogs.mapToPair(log -> new Tuple2<>(log.getIpAddress(), 1L))
        .reduceByKey(SUM_REDUCER)
        .filter(tuple -> tuple._2() > 10)
        .map(Tuple2::_1)
        .take(100);
System.out.println(String.format("IPAddresses > 10 times: %s", ipAddresses));
```

Last, let's calculate the top endpoints requested in this log file. We define an inner class, `ValueComparator` to help with that. This function tells us, given two tuples, which one is first in ordering. The key of the tuple is ignored, and ordering is based just on the values.

```
private static class ValueComparator<K, V>
   implements Comparator<Tuple2<K, V>>, Serializable {
  private Comparator<V> comparator;

  public ValueComparator(Comparator<V> comparator) {
    this.comparator = comparator;
  }

  @Override
  public int compare(Tuple2<K, V> o1, Tuple2<K, V> o2) {
    return comparator.compare(o1._2(), o2._2());
  }
}
```

Then, we can use the `ValueComparator` with the `top` action to compute the top endpoints accessed on this server according to how many times the endpoint was accessed.

```
List<Tuple2<String, Long>> topEndpoints = accessLogs
    .mapToPair(log -> new Tuple2<>(log.getEndpoint(), 1L))
    .reduceByKey(SUM_REDUCER)
    .top(10, new ValueComparator<>(Comparator.<Long>naturalOrder()));
System.out.println("Top Endpoints: " + topEndpoints);
```

These code snippets are from LogAnalyzer.java. Now that we've walked through the code, try running that example. See the README for language specific instructions for building and running.

Spark SQL

You should go through the Spark SQL Guide before beginning this section.

This section requires an additioal dependency on Spark SQL:

```
<dependency> <!-- Spark SQL -->
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.10</artifactId>
    <version>1.1.0</version>
</dependency>
```

For those of you who are familiar with SQL, the same statistics we calculated in the previous example can be done using Spark SQL rather than calling Spark transformations and actions directly. We walk through how to do that here.

First, we need to create a SQL Spark context. Note how we create one Spark Context, and then use that to instantiate different flavors of Spark contexts. You should not initialize multiple Spark contexts from the SparkConf in one process.

```
public class LogAnalyzerSQL {
  public static void main(String[] args) {
    // Create the spark context.
    SparkConf conf = new SparkConf().setAppName("Log Analyzer SQL");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaSQLContext sqlContext = new JavaSQLContext(sc);

    if (args.length == 0) {
      System.out.println("Must specify an access logs file.");
      System.exit(-1);
    }
    String logFile = args[0];
    JavaRDD<ApacheAccessLog> accessLogs = sc.textFile(logFile)
        .map(ApacheAccessLog::parseFromLogLine);

    // TODO: Insert code for computing log stats.

    sc.stop();
  }
}
```

Next, we need a way to register our logs data into a table. In Java, Spark SQL can infer the table schema on a standard Java POJO - with getters and setters as we've done with ApacheAccessLog.java. (Note: if you are using a different language besides Java, there is a different way for Spark to infer the table schema. The examples in this directory work out of the box. Or you can also refer to the Spark SQL Guide on Data Sources for more details.)

```
JavaSchemaRDD schemaRDD = sqlContext.applySchema(accessLogs,
    ApacheAccessLog.class);
schemaRDD.registerTempTable("logs");
sqlContext.sqlContext().cacheTable("logs");
```

Now, we are ready to start running some SQL queries on our table. Here's the code to compute the identical statistics in the previous section - it should look very familiar for those of you who know SQL:

```
// Calculate statistics based on the content size.
Tuple4<Long, Long, Long, Long> contentSizeStats =
    sqlContext.sql("SELECT SUM(contentSize), COUNT(*), MIN(contentSize), MAX(contentSize) FROM logs")
        .map(row -> new Tuple4<>(row.getLong(0), row.getLong(1), row.getLong(2), row.getLong(3)))
        .first();
```

```
System.out.println(String.format("Content Size Avg: %s, Min: %s, Max: %s",
    contentSizeStats._1() / contentSizeStats._2(),
    contentSizeStats._3(),
    contentSizeStats._4()));

// Compute Response Code to Count.
// Note the use of "LIMIT 1000" since the number of responseCodes
// can potentially be too large to fit in memory.
List<Tuple2<Integer, Long>> responseCodeToCount = sqlContext
    .sql("SELECT responseCode, COUNT(*) FROM logs GROUP BY responseCode LIMIT 1000")
    .mapToPair(row -> new Tuple2<>(row.getInt(0), row.getLong(1)));
System.out.println(String.format("Response code counts: %s", responseCodeToCount))
    .collect();

// Any IPAddress that has accessed the server more than 10 times.
List<String> ipAddresses = sqlContext
    .sql("SELECT ipAddress, COUNT(*) AS total FROM logs GROUP BY ipAddress HAVING total > 10 LIMIT 100")
    .map(row -> row.getString(0))
    .collect();
System.out.println(String.format("IPAddresses > 10 times: %s", ipAddresses));

// Top Endpoints.
List<Tuple2<String, Long>> topEndpoints = sqlContext
    .sql("SELECT endpoint, COUNT(*) AS total FROM logs GROUP BY endpoint ORDER BY total DESC LIMIT 10")
    .map(row -> new Tuple2<>(row.getString(0), row.getLong(1)))
    .collect();
System.out.println(String.format("Top Endpoints: %s", topEndpoints));
```

Note that the default SQL dialect does not allow using reserved keyworks as alias names. In other words, `SELECT COUNT(*) AS count` will cause errors, but `SELECT COUNT(*) AS the_count` runs fine. If you use the HiveQL parser though, then you should be able to use anything as an identifier.

Try running LogAnalyzerSQL.java now.

Spark Streaming

Go through the Spark Streaming Programming Guide before beginning this section. In particular, it covers the concept of DStreams.

This section requires another dependency on the Spark Streaming library:

```
<dependency> <!-- Spark Streaming -->
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.10</artifactId>
    <version>1.1.0</version>
</dependency>
```

The earlier examples demonstrates how to compute statistics on an existing log file - but not how to do realtime monitoring of logs. Spark Streaming enables that functionality.

To run the streaming examples, you will `tail` a log file into `netcat` to send to Spark. This is not the ideal way to get data into Spark in a production system, but is an easy workaround for a first Spark Streaming example. We will cover best practices for how to import data for Spark Streaming in Chapter 2.

In a terminal window, just run this command on a logfile which you will append to:

```
% tail -f [[YOUR_LOG_FILE]] | nc -lk 9999
```

If you don't have a live log file that is being updated on the fly, you can add lines manually with the included data file or another your own log file:

```
% cat ../../data/apache.accesslog >> [[YOUR_LOG_FILE]]
```

When data is streamed into Spark, there are two common use cases covered:

1. Windowed Calculations means that you only care about data received in the last N amount of time. When monitoring your web servers, perhaps you only care about what has happened in the last hour.
    - Spark Streaming conveniently splits the input data into the desired time windows for easy processing, using the `window` function of the streaming library.
    - The `forEachRDD` function allows you to access the RDD's created each time interval.
2. Cumulative Calculations means that you want to keep cumulative statistics, while streaming in new data to refresh those statistics. In that case, you need to maintain the state for those statistics.
    - The Spark Streaming library has some convenient functions for maintaining state to support this use case, `updateStateByKey`.
3. Reusing code from Batching covers how to organize business logic code from the batch examples so that code can be reused in Spark Streaming.
    - The Spark Streaming library has `transform` functions which allow you to apply arbitrary RDD-to-RDD functions, and thus to reuse code from the batch mode of Spark.

Windowed Calculations: window()

A typical use case for log analysis is monitoring a web server, in which case you may only be interested in what's happened for the last one hour of time and want those statistics to refresh every minute. One hour is the *window length*, while one minute is the *slide interval*. In this example, we use a window length of 30 seconds and a slide interval of 10 seconds as a comfortable choice for development.

The windows feature of Spark Streaming makes it very easy to compute stats for a window of time, using the `window` function.

The first step is to initalize the SparkConf and context objects - in particular a streaming context. Note how only one SparkContext is created from the conf and the streaming and sql contexts are created from those. Next, the main body should be written. Finally, the example calls `start()` on the streaming context, and `awaitTermination()` to keep the streaming context running and accepting streaming input.

```
public class LogAnalyzerStreamingSQL {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("Log Analyzer Streaming SQL");

    // Note: Only one Spark Context is created from the conf, the rest
    //       are created from the original Spark context.
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaStreamingContext jssc = new JavaStreamingContext(sc,
        SLIDE_INTERVAL);  // This sets the update window to be every 10 seconds.
    JavaSQLContext sqlContext = new JavaSQLContext(sc);

    // TODO: Insert code here to process logs.

    // Start the streaming server.
    jssc.start();              // Start the computation
    jssc.awaitTermination();   // Wait for the computation to terminate
  }
}
```

The first step of the main body is to create a DStream from reading the socket.

```
JavaReceiverInputDStream<String> logDataDStream =
    jssc.socketTextStream("localhost", 9999);
```

Next, call the `map` transformation to convert the logDataDStream into a ApacheAccessLog DStream.

```
JavaDStream<ApacheAccessLog> accessLogDStream =
    logDataDStream.map(ApacheAccessLog::parseFromLogLine).cache();
```

Next, call `window` on the accessLogDStream to create a windowed DStream. The `window` function nicely packages the input data that is being streamed into RDDs containing a window length of data, and creates a new RDD every SLIDE_INTERVAL of time.

```
JavaDStream<ApacheAccessLog> windowDStream =
    accessLogDStream.window(WINDOW_LENGTH, SLIDE_INTERVAL);
```

Then call `foreachRDD` on the windowDStream. The function passed into `forEachRDD` is called on each new RDD in the windowDStream as the RDD is created, so every *slide_interval*. The RDD passed into the function contains all the input for

the last *window_length* of time. Now that there is an RDD of ApacheAccessLogs, simply reuse code from either two batch examples (regular or SQL). In this example, the code was just copied and pasted, but you could refactor this code into one place nicely for reuse in your production code base - you can reuse all your batch processing code for streaming!

```
windowDStream.foreachRDD(accessLogs -> {
  if (accessLogs.count() == 0) {
    System.out.println("No access logs in this time interval");
    return null;
  }

  // Insert code verbatim from LogAnalyzer.java or LogAnalyzerSQL.java here.

  // Calculate statistics based on the content size.
  JavaRDD<Long> contentSizes =
      accessLogs.map(ApacheAccessLog::getContentSize).cache();
  System.out.println(String.format("Content Size Avg: %s, Min: %s, Max: %s",
      contentSizes.reduce(SUM_REDUCER) / contentSizes.count(),
      contentSizes.min(Comparator.naturalOrder()),
      contentSizes.max(Comparator.naturalOrder())));

   //...Won't copy the rest here...
}
```

Now that we've walked through the code, run LogAnalyzerStreaming.java and/or LogAnalyzerStreamingSQL.java now. Use the `cat` command as explained before to add data to the log file periodically once you have your program up.

Cumulative Calculations: updateStateByKey()

To keep track of the log statistics for all of time, state must be maintained between processing RDD's in a DStream.

To maintain state for key-pair values, the data may be too big to fit in memory on one machine - Spark Streaming can maintain the state for you. To do that, call the `updateStateByKey` function of the Spark Streaming library.

First, in order to use `updateStateByKey`, checkpointing must be enabled on the streaming context. To do that, just call `checkpoint` on the streaming context with a directory to write the checkpoint data. Here is part of the main function of a streaming application that will save state for all of time:

```
public class LogAnalyzerStreamingTotal {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("Log Analyzer Streaming Total");
    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaStreamingContext jssc = new JavaStreamingContext(sc,
        new Duration(10000));  // This sets the update window to be every 10 seconds.

    // Checkpointing must be enabled to use the updateStateByKey function.
    jssc.checkpoint("/tmp/log-analyzer-streaming");

    // TODO: Insert code for computing log stats.

    // Start the streaming server.
    jssc.start();              // Start the computation
    jssc.awaitTermination();   // Wait for the computation to terminate
```

To compute the content size statistics, simply use static variables to save the current running sum, count, min and max of the content sizes.

```
// These static variables stores the running content size values.
private static final AtomicLong runningCount = new AtomicLong(0);
private static final AtomicLong runningSum = new AtomicLong(0);
private static final AtomicLong runningMin = new AtomicLong(Long.MAX_VALUE);
private static final AtomicLong runningMax = new AtomicLong(Long.MIN_VALUE);
```

To update those values, first call map on the AccessLogDStream to retrieve a contentSizeDStream. Then just update the values for the static variables by calling foreachRDD on the contentSizeDstream, and calling actions on the RDD:

```
JavaDStream<Long> contentSizeDStream =
    accessLogDStream.map(ApacheAccessLog::getContentSize).cache();
contentSizeDStream.foreachRDD(rdd -> {
  if (rdd.count() > 0) {
    runningSum.getAndAdd(rdd.reduce(SUM_REDUCER));
    runningCount.getAndAdd(rdd.count());
    runningMin.set(Math.min(runningMin.get(), rdd.min(Comparator.naturalOrder())));
    runningMax.set(Math.max(runningMax.get(), rdd.max(Comparator.naturalOrder())));
    System.out.print("Content Size Avg: " +  runningSum.get() / runningCount.get());
    System.out.print(", Min: " + runningMin.get());
    System.out.println(", Max: " + runningMax.get());
  }
  return null;
});
```

For the other statistics, since they make use of key value pairs, static variables can't be used anymore. The amount of state that needs to be maintained is potentially too big to fit in memory. So for those stats, we'll make use of `updateStateByKey` so Spark streaming will maintain a value for every key in our dataset.

But before we can call `updateStateByKey`, we need to create a function to pass into it. `updateStateByKey` takes in a different reduce function. While our previous sum reducer just took in two values and output their sum, this reduce function takes in a current value and an iterator of values, and outputs one new value.

```
private static Function2<List<Long>, Optional<Long>, Optional<Long>>
   COMPUTE_RUNNING_SUM = (nums, current) -> {
     long sum = current.or(0L);
     for (long i : nums) {
       sum += i;
     }
     return Optional.of(sum);
   };
```

Finally, we can compute the keyed statistics for all of time with this code:

```
// Compute Response Code to Count.
// Note the use of updateStateByKey.
JavaPairDStream<Integer, Long> responseCodeCountDStream = accessLogDStream
    .mapToPair(s -> new Tuple2<>(s.getResponseCode(), 1L))
    .reduceByKey(SUM_REDUCER)
    .updateStateByKey(COMPUTE_RUNNING_SUM);
responseCodeCountDStream.foreachRDD(rdd -> {
  System.out.println("Response code counts: " + rdd.take(100));
  return null;
});

// A DStream of ipAddresses accessed > 10 times.
JavaDStream<String> ipAddressesDStream = accessLogDStream
    .mapToPair(s -> new Tuple2<>(s.getIpAddress(), 1L))
    .reduceByKey(SUM_REDUCER)
    .updateStateByKey(COMPUTE_RUNNING_SUM)
    .filter(tuple -> tuple._2() > 10)
    .map(Tuple2::_1);
ipAddressesDStream.foreachRDD(rdd -> {
  List<String> ipAddresses = rdd.take(100);
  System.out.println("All IPAddresses > 10 times: " + ipAddresses);
  return null;
});

// A DStream of endpoint to count.
JavaPairDStream<String, Long> endpointCountsDStream = accessLogDStream
    .mapToPair(s -> new Tuple2<>(s.getEndpoint(), 1L))
    .reduceByKey(SUM_REDUCER)
    .updateStateByKey(COMPUTE_RUNNING_SUM);
endpointCountsDStream.foreachRDD(rdd -> {
  List<Tuple2<String, Long>> topEndpoints =
      rdd.takeOrdered(10, new ValueComparator<>(Comparator.<Long>naturalOrder()));
  System.out.println("Top Endpoints: " + topEndpoints);
  return null;
});
```

Run LogAnalyzerStreamingTotal.java now for yourself.

Reusing Code from Batching: transform()

As you may have noticed, while the functions you called on a DStream are named the same as those you called on an RDD in the batch example, they are not the same methods, and it may not be clear how to reuse the code from the batch examples. In this section, we refactor the code from the batch examples and show how to reuse it here.

DStreams have `transform` functions which allows you to call any arbitrary RDD to RDD functions to RDD's in the DStream. The `transform` functions are perfect for reusing any RDD to RDD functions that you may have written in batch code and want to port over to streaming. Let's look at some code to illustrate this point.

Let's say we have separated out a function, `responseCodeCount` from our batch example that can compute the response code count given the apache access logs RDD:

```
public static JavaPairRDD<Integer, Long> responseCodeCount(
   JavaRDD<ApacheAccessLog> accessLogRDD) {
  return accessLogRDD
     .mapToPair(s -> new Tuple2<>(s.getResponseCode(), 1L))
     .reduceByKey(SUM_REDUCER);
}
```

The responseCodeCountDStream can be created by calling `transformToPair` with the `responseCodeCount` function to the accessLogDStream. Then, you can finish up by calling `updateStateByKey` to keep a running count of the response codes for all of time, and use `forEachRDD` to print the values out:

```
// Compute Response Code to Count.
// Notice the user transformToPair to produce the a DStream of
// response code counts, and then updateStateByKey to accumulate
// the response code counts for all of time.
JavaPairDStream<Integer, Long> responseCodeCountDStream = accessLogDStream
   .transformToPair(LogAnalyzerStreamingTotalRefactored::responseCodeCount);
JavaPairDStream<Integer, Long> cumulativeResponseCodeCountDStream =
   responseCodeCountDStream.updateStateByKey(COMPUTE_RUNNING_SUM);
cumulativeResponseCodeCountDStream.foreachRDD(rdd -> {
  System.out.println("Response code counts: " + rdd.take(100));
  return null;
});
```

It is possible to combine `transform` functions before and after an `updateStateByKey` as well:

```
// A DStream of ipAddresses accessed > 10 times.
JavaDStream<String> ipAddressesDStream = accessLogDStream
   .transformToPair(LogAnalyzerStreamingTotalRefactored::ipAddressCount)
   .updateStateByKey(COMPUTE_RUNNING_SUM)
   .transform(LogAnalyzerStreamingTotalRefactored::filterIPAddress);
ipAddressesDStream.foreachRDD(rdd -> {
  List<String> ipAddresses = rdd.take(100);
  System.out.println("All IPAddresses > 10 times: " + ipAddresses);
  return null;
});
```

Take a closer look at LogAnalyzerStreamingTotalRefactored.java now to see how that code has been refactored to reuse code from the batch example.

Section 2: Importing Data

In the last section we covered how to get started with Spark for log analysis, but in those examples, data was just pulled in from a local file and the statistics were printed to standard out. In this chapter, we cover techniques for loading and exporting data that is suitable for a production system. In particular, the techniques must scale to handle large production volumes of logs.

To scale, Apache Spark is meant to be deployed on a cluster of machines. Read the Spark Cluster Overview Guide, so that you understand the different between the Spark driver vs. the executor nodes.

While you could continue running the examples in local mode, it is recommended that you set up a Spark cluster to run the remaining examples on and get practice working with the cluster - such as familiarizing yourself with the web interface of the cluster. You can run a small cluster on your local machine by following the instructions for Spark Standalone Mode. Optionally, if you have access to more machines - such as on AWS or your organization has its own datacenters, consult the cluster overview guide to do that.

Once you get a Spark cluster up:

- Use spark-submit to run your jobs rather than using the JVM parameter. Run one of the examples from the previous chapter to check your set up.
- Poke around and familiarize with the web interfaces for Spark. It's at http://localhost:8080 if you set up a local cluster.

There are two ways to import data into Spark:

1. Batch Data Import - if you are loading a dataset all at once.
2. Streaming Data Import - if you wish to continuously stream data into Spark.

Batch Data Import

This section covers batch importing data into Apache Spark, such as seen in the non-streaming examples from Chapter 1. Those examples load data from files all at once into one RDD, processes that RDD, the job completes, and the program exits. In a production system, you could set up a cron job to kick off a batch job each night to process the last day's worth of log files and then publish statistics for the last day.

- Importing From Files covers caveats when importing data from files.
- Importing from Databases links to examples of reading data from databases.

Importing from Files

To support batch import of data on a Spark cluster, the data needs to be accessible by all machines on the cluster. Files that are only accessible on one worker machine and cannot be read by the others will cause failures.

If you have a small dataset that can fit on one machine, you could manually copy your files onto all the nodes on your Spark cluster, perhaps using `rsync` to make that easier.

NFS or some other network file system makes sure all your machines can access the same files without requiring you to copy the files around. But NFS isn't fault tolerant to machine failures and if your dataset is too big to fit on one NFS volume - you'd have to store the data on multiple volumes and figure out which volume a particular file is on - which could get cumbersome.

HDFS and **S3** are great file systems for massive datasets - built to store a lot of data and give all the machines on the cluster access to those files, while still being fault tolerant. We give a few more tips on running Spark with these file systems since they are recommended.

- S3 is an Amazon AWS solution for storing files in the cloud, easily accessible to anyone who signs up for an account.
- HDFS is a distributed file system that is part of Hadoop and can be installed on your own datacenters.

The good news is that regardless of which of these file systems you choose, you can run the same code to read from them - these file systems are all "Hadoop compatible" file systems.

In this section, you should try running LogAnalyzerBatchImport.java on any files on your file system of choice. There is nothing new in this code - it's just a refactor of the First Log Analyzer from Chapter One. Try passing in "*" or "?" for the textFile path, and Spark will read in all the files that match that pattern to create the RDD.

S3

S3 is Amazon Web Services's solution for storing large files in the cloud. On a production system, you want your Amazon EC2 compute nodes on the same zone as your S3 files for speed as well as cost reasons. While S3 files can be read from other machines, it would take a long time and be expensive (Amazon S3 data transfer prices differ if you read data within AWS vs. to somewhere else on the internet).

See running Spark on EC2 if you want to launch a Spark cluster on AWS - charges apply.

If you choose to run this example with a local Spark cluster on your machine rather than EC2 compute nodes to read the files in S3, use a small data input source!

1. Sign up for an Amazon Web Services Account.
2. Load example log files to s3.
    - Log into the AWS console for S3
    - Create an S3 bucket.
    - Upload a couple of example log files to that bucket.
    - Your files will be at the path: s3n://YOUR_BUCKET_NAME/YOUR_LOGFILE.log
3. Configure your security credentials for AWS:
    - Create and download your security credentials
    - Set the environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY to the correct values on all machines on your cluster. These can also be set in your SparkContext object programmatically like this:

```
jssc.hadoopConfiguration().set("fs.s3n.awsAccessKeyId", YOUR_ACCESS_KEY)
jssc.hadoopConfiguration().set("fs.s3n.awsSecretAccessKey", YOUR_SECRET_KEY)
```

Now, run LogAnalyzerBatchImport.java passing in the s3n path to your files.

HDFS

HDFS is a file system that is meant for storing large data sets and being fault tolerant. In a production system, your Spark cluster should ideally be on the same machines as your Hadoop cluster to make it easy to read files. The Spark binary you run on your clusters must be compiled with the same HDFS version as the one you wish to use.

There are many ways to install HDFS, but heading to the Hadoop homepage is one way to get started and run hdfs locally on your machine.

Run LogAnalyzerBatchImport.java on any file pattern on your hdfs directory.

Reading from Databases

Most likely, you aren't going to be storing your logs data in a database (that is likely too expensive), but there may be other data you want to input to Spark that is stored in a database. Perhaps that data can be joined with the logs to provide more information.

The same way file systems have evolved over time to scale, so have databases.

A simple database to begin with is a single database - SQL databases are quite common. When that fills, one option is to buy a larger machine for the database. The price of these larger machines gets increasingly expensive (even price per unit of storage) and it is eventually no longer possible to buy a machine big enough at some point. A common choice then is to switch to sharded databases. With that option, application level code is written to determine on which database shard a piece of data should be read or written to.

To read data in from a SQL database, the JdbcRDD is one option for a moderate amount of data:

- https://spark.apache.org/docs/0.8.1/api/core/org/apache/spark/rdd/JdbcRDD.html

Recently, there has been a movement in the database world towards **NoSQL** or **Key-Value** databases that were designed to scale. For these databases, it's usually transparent to the application developer that the underlying database stores data on multiple machines. **Cassandra** is one very popular NoSQL database.

To read data from Cassandra into Spark, see the Spark Cassandra Connector:

- https://github.com/datastax/spark-cassandra-connector

If you use a different database, Spark may have a built-in library for importing from that database, but more often 3rd parties offer Spark integration - so search for that.

As usual, reading a small amount of data from a database is much easier than reading a ton of data. It's important to understand your database and Spark's distributed programming model in order to write optimal code for importing a very large dataset.

Streaming Data Import

This section covers importing data for streaming. The streaming example in the previous chapter received data through a single socket - which is not a scalable solution. In a real production system, there are many servers continuously writing logs, and we want to process all of those files. This section contains scalable solutions for data import. Since streaming is now used, there is no longer the need for a nightly batch job to process logs, but instead - this logs processing program can be long-lived - continuously receiving new logs data, processing the data, and computing log stats.

1. Built In Methods for Streaming Import
2. Kafka

Built In Methods for Streaming Import

The StreamingContext has many built in methods for importing data to streaming. `socketTextStream` was introduced in the previous chapter, and `textFileStream` is introduced here. The `textFileStream` method monitors any Hadoop-compatible filesystem directory for new files and when it detects a new file - reads it into Spark Streaming. Just replace the call to `socketTextStream` with `textFileStream`, and pass in the directory to monitor for log files.

```
// This methods monitors a directory for new files
// to read in for streaming.
JavaDStream<String> logData = jssc.textFileStream(directory);
```

Try running LogAnalyzerStreamingImportDirectory.java by specifying a directory. You'll also need to drop or copy some new log files into that directory while the program is running to see the calculated values update.

There are more built-in input methods for streaming - check them out in the reference API documents for the StreamingContext.

Kafka

While the previous example picks up new log files right away - the log files aren't copied over until a long time after the HTTP requests in the logs actually occurred. While that enables auto-refresh of log data, that's still not realtime. To get realtime logs processing, we need a way to send over log lines immediately. Kafka is a high-throughput distributed message system that is perfect for that use case. Spark contains an external module importing data from Kafka.

Here is some useful documentation to set up Kafka for Spark Streaming:

- Kafka Documentation
- KafkaUtils class in the external module of the Spark project - This is the external module that has been written that imports data from Kafka into Spark Streaming.
- Spark Streaming Example of using Kafka - This is an example that demonstrates how to call KafkaUtils.

Exporting Data out of Spark

This section contains methods for exporting data out of Spark into systems. First, you'll have to figure out if your output data is small (meaning can fit on memory on one machine) or large (too big to fit into memory on one machine). Consult these two sections based on your use case.

- Small Datasets - If you have a small dataset, you can call an action on this dataset to retrieve objects in memory on the driver program, and then write those objects out any way you want.
- Large Datasets - For a large dataset, it's important to remember that this dataset is too large to fit in memory on the driver program. In that case, you can either call Spark to write the data to files directly from the Spark workers or you can implement your own custom solution.

Exporting Small Datasets

If the data you are exporting out of Spark is small, you can just use an action to convert the RDD into objects in memory on the driver program, and then write that output directly to any data storage solution of your choosing. You may remember that we called the `take(N)` action where N is some finite number instead of the `collect()` action to ensure the output fits in memory - no matter how big the input data set may be - this is good practice. This section walks through example code where you'll write the log statistics to a file.

It may not be that useful to have these stats output to a file - in practice, you might write these statistics to a database for your presentation layer to access.

```
LogStatistics logStatistics = logAnalyzerRDD.processRdd(accessLogs);

String outputFile = args[1];
Writer out = new BufferedWriter(
    new OutputStreamWriter(new FileOutputStream(outputFile)));

Tuple4<Long, Long, Long, Long> contentSizeStats =
    logStatistics.getContentSizeStats();
out.write(String.format("Content Size Avg: %s, Min: %s, Max: %s\n",
    contentSizeStats._1() / contentSizeStats._2(),
    contentSizeStats._3(),
    contentSizeStats._4()));

List<Tuple2<Integer, Long>> responseCodeToCount =
    logStatistics.getResponseCodeToCount();
out.write(String.format("Response code counts: %s\n", responseCodeToCount));

List<String> ipAddresses = logStatistics.getIpAddresses();
out.write(String.format("IPAddresses > 10 times: %s\n", ipAddresses));

List<Tuple2<String, Long>> topEndpoints = logStatistics.getTopEndpoints();
out.write(String.format("Top Endpoints: %s\n", topEndpoints));

out.close();
```

Now, run LogAnalyzerExportSmallData.java. Try modifying it to write to a database of your own choosing.

Exporting Large Datasets

If you are exporting a very large dataset, you can't call `collect()` or a similar action to read all the data from the RDD onto the single driver program - that could trigger out of memory problems. Instead, you have to be careful about saving a large RDD. See these two sections for more information.

- Save the RDD to Files - There are built in methods in Spark for saving a large RDD to files.
- Save the RDD to a Database - This section contains recommended best practices for saving a large RDD to a database.

Save the RDD to files

RDD's have some built in methods for saving them to disk. Once in files, many of the Hadoop databases can bulk load in data directly from files, as long as they are in a specific format.

In the following code example, we demonstrate the simple `.saveAsTextFile()` method. This will write the data to simple text files where the `.toString()` method is called on each RDD element and one element is written per line. The number of files output is equal to the the number of partitions of the RDD being saved. In this sample, the RDD is repartitioned to control the number of output files.

```
public class LogAnalyzerExportRDD {
  // Optionally modify this based as makes sense for your dataset.
  public static final int NUM_PARTITIONS = 2;

  public static void main(String[] args) throws IOException {
    // Create the spark context.
    SparkConf conf = new SparkConf().setAppName("Log Analyzer SQL");
    JavaSparkContext sc = new JavaSparkContext(conf);

    if (args.length < 2) {
      System.out.println("Must specify an access logs file and an output file.");
      System.exit(-1);
    }
    String inputFile = args[0];
    String outputDirectory = args[1];
    JavaRDD<ApacheAccessLog> accessLogs = sc.textFile(inputFile)
        .map(ApacheAccessLog::parseFromLogLine)
        .repartition(NUM_PARTITIONS); // Optionally, change this.

    accessLogs.saveAsTextFile(outputDirectory);

    sc.stop();
  }
}
```

Run LogAnalyzerExportRDD.java now. Notice that the number of files output is the same as the number of partitionds of the RDD.

Refer to the API documentation for other built in methods for saving to file. There are different built in methods for saving RDD's to files in various formats, so skim the whole RDD package to see if there is something to suit your needs.

Sqoop is a very useful tool that can import Hadoop files into various databases, and is thus very useful to use for getting the data written into files from Spark into your production database.

Save an RDD to a Database

You can write your own custom writer and call a transform on your RDD to write each element to a database of your choice, but there's a lot of ways to write something that looks like it would work, but does not work well in a distributed environment. Here are some things to watch out for:

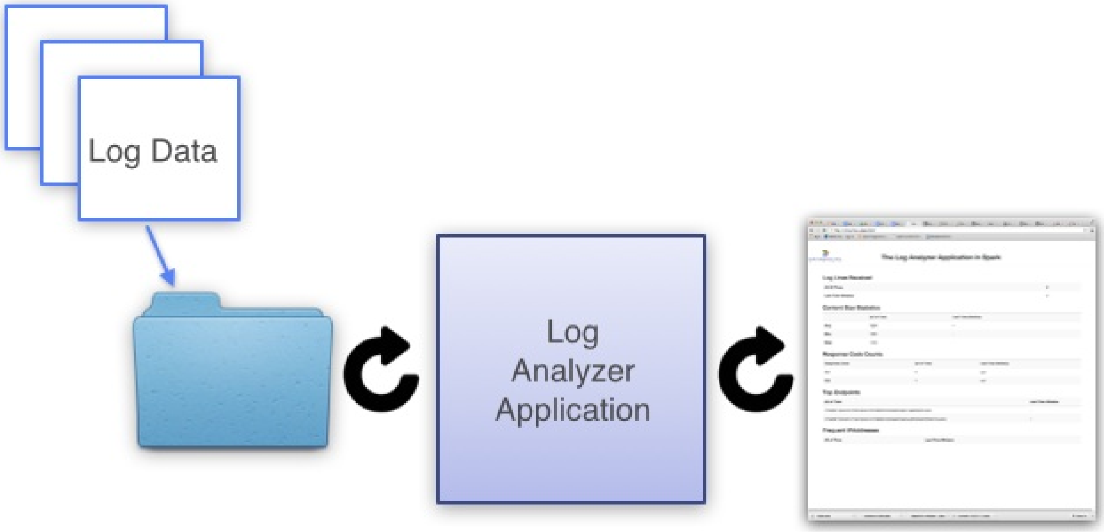
- A common naive mistake is to open a connection on the Spark driver program, and then try to use that connection on the Spark workers. The connection should be opened on the Spark worker, such as by calling `forEachPartition` and opening the connection inside that function.
- Use partitioning to control the parallelism for writing to your data storage. Your data storage may not support too many concurrent connections.
- Use batching for writing out multiple objects at a time if batching is optimal for your data storage.
- Make sure your write mechanism is resilient to failures. Writing out a very large dataset can take a long time, which increases the chance something can go wrong - a network failure, etc.
- Consider utilizing a static pool of database connections on your Spark workers.
- If you are writing to a sharded data storage, partition your RDD to match your sharding strategy. That way each of your Spark workers only connects to one database shard, rather than each Spark worker connecting to every database shard.

Be cautious when writing out so much data, and make sure you understand the distributed nature of Spark!

Logs Analyzer Application

This directory contains code from the chapters, assembled together to form a sample logs analyzer application. Other libraries that are not discussed have been used to make this a more finished application. These are the features of our MVP (minimal viable product) logs analyzer application:

- Reads in new log files from a directory and inputs those new files into Spark Streaming.
- Compute stats on the logs using Spark - stats for the last 30 seconds are calculated as well as for all of time.
- Write the calculated stats to an html file on the local file system that gets refreshed on a set interval.

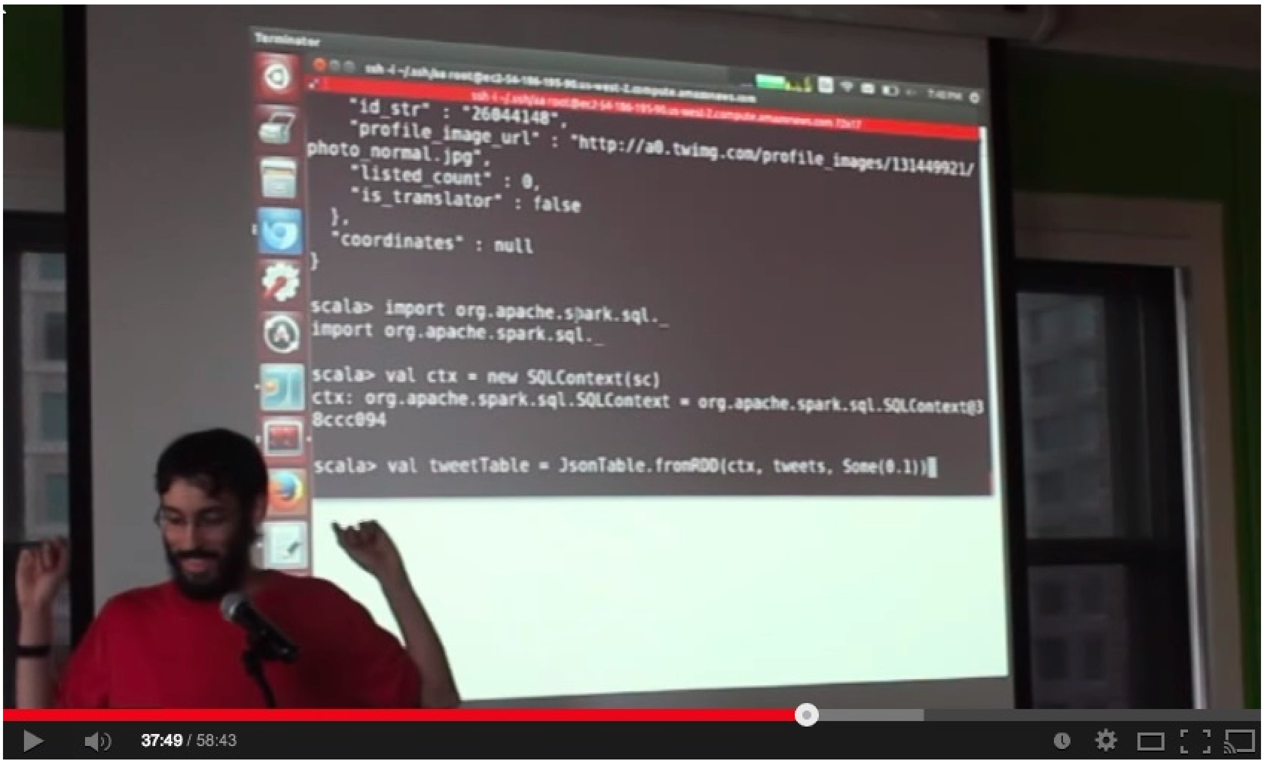


You can use this simple application as a skeleton and combine features from the chapters to produce your own custom logs analysis application. The main class is LogAnalyzerAppMain.java.

Twitter Streaming Language Classifier

In this reference application, we show how you can use Apache Spark for training a language classifier - replacing a whole suite of tools you may be currently using.

This reference application was demo-ed at a meetup which is taped here - the link skips straight to demo time, but the talk before that is useful too:



Here are 5 typical stages for creating a production-ready classifier. Often, each stage is done with a different set of tools and even by different engineering teams:

1. Scrape/collect a dataset.
2. Clean and explore the data, doing feature extraction.
3. Build a model on the data and iterate/improve it.
4. Improve the model using more and more data, perhaps upgrading your infrastructure to support building larger models. (Such as migrating over to Hadoop.)
5. Apply the model in real time.

Spark can be used for all of the above and simple to use for all these purposes. We've chosen to break up the language classifier into 3 parts with one simple Spark program to accomplish each part:

1. Collect a Dataset of Tweets - Spark Streaming is used to collect a dataset of tweets and write them out to files.
2. Examine the Tweets and Train a Model - Spark SQL is used to examine the dataset of Tweets. Then Spark MLLib is used to apply the K-Means algorithm to train a model on the data.
3. Apply the Model in Real-time - Spark Streaming and Spark MLLib are used to filter a live stream of Tweets for those that match the specified cluster.

Part 1: Collect a Dataset of Tweets

Spark Streaming is used to collect tweets as the dataset. The tweets are written out in JSON format, one tweet per line. A file of tweets is written every time interval until at least the desired number of tweets is collected.

See Collect.scala for the full code. We'll walk through some of the interesting bits now.

Collect.scala takes in the following argument list:

1. *outputDirectory* - the output directory for writing the tweets. The files will be named 'part-%05d'
2. *numTweetsToCollect* - this is the minimum number of tweets to collect before the program exits.
3. *intervalInSeconds* - write out a new set of tweets every interval.
4. *partitionsEachInterval* - this is used to control the number of output files written for each interval

Collect.scala will also require Twitter API Credentials. If you have never signed up for Twitter Api Credentials, follow these steps here. The Twitter credentials are passed in through command line flags.

Below is a snippet of the actual code in Collect.scala. The code calls TwitterUtils in the Spark Streaming Twitter library to get a DStream of tweets. Then, map is called to convert the tweets to JSON format. Finally, call for each RDD on the DStream. This example repartitions the RDD to write out so that you can control the number of output files.

```
 val tweetStream = TwitterUtils.createStream(ssc, Utils.getAuth)
  .map(gson.toJson(_))

tweetStream.foreachRDD((rdd, time) => {
  val count = rdd.count()
  if (count > 0) {
    val outputRDD = rdd.repartition(partitionsEachInterval)
    outputRDD.saveAsTextFile(
      outputDirectory + "/tweets_" + time.milliseconds.toString)
    numTweetsCollected += count
    if (numTweetsCollected > numTweetsToCollect) {
      System.exit(0)
    }
  }
})
```

Run Collect.scala yourself to collect a dataset of tweets:

```
 %  ${YOUR_SPARK_HOME}/bin/spark-submit \
     --class "com.databricks.apps.twitter_classifier.Collect" \
     --master ${YOUR_SPARK_MASTER:-local[4]} \
     target/scala-2.10/spark-twitter-lang-classifier-assembly-1.0.jar \
     ${YOUR_OUTPUT_DIR:-/tmp/tweets} \
     ${NUM_TWEETS_TO_COLLECT:-10000} \
     ${OUTPUT_FILE_INTERVAL_IN_SECS:-10} \
     ${OUTPUT_FILE_PARTITIONS_EACH_INTERVAL:-1} \
     --consumerKey ${YOUR_TWITTER_CONSUMER_KEY} \
     --consumerSecret ${YOUR_TWITTER_CONSUMER_SECRET} \
     --accessToken ${YOUR_TWITTER_ACCESS_TOKEN}  \
     --accessTokenSecret ${YOUR_TWITTER_ACCESS_SECRET}
```

Part 2: Examine Tweets and Train a Model

The second program examines the data found in tweets and trains a language classifier using K-Means clustering on the tweets:

- Examine - Spark SQL is used to gather data about the tweets -- to look at a few of them, and to count the total number of tweets for the most common languages of the user.
- Train - Spark MLLib is used for applying the K-Means algorithm for clustering the tweets. The number of clusters and the number of iterations of algorithm are configurable. After training the model, some sample tweets from the different clusters are shown.

See here for the command to run part 2.

Examine with Spark SQL

Spark SQL can be used to examine data based on the tweets. Below are some relevant code snippets from ExamineAndTrain.scala.

First, here is code to pretty print 5 sample tweets so that they are more human-readable.

```
val tweets = sc.textFile(tweetInput)
for (tweet <- tweets.take(5)) {
  println(gson.toJson(jsonParser.parse(tweet)))
}
```

Spark SQL can load JSON files and infer the schema based on that data. Here is the code to load the json files, register the data in the temp table called "tweetTable" and print out the schema based on that.

```
val tweetTable = sqlContext.jsonFile(tweetInput)
tweetTable.registerTempTable("tweetTable")
tweetTable.printSchema()
```

Now, look at the text of 10 sample tweets.

```
sqlContext.sql(
    "SELECT text FROM tweetTable LIMIT 10")
    .collect().foreach(println)
```

View the user language, user name, and text for 10 sample tweets.

```
sqlContext.sql(
    "SELECT user.lang, user.name, text FROM tweetTable LIMIT 10")
    .collect().foreach(println)
```

Finally, show the count of tweets by user language. This can help determine the number of clusters that is ideal for this dataset of tweets.

```
sqlContext.sql(
    "SELECT user.lang, COUNT(*) as cnt FROM tweetTable " +
    "GROUP BY user.lang ORDER BY cnt DESC limit 1000")
    .collect.foreach(println)
```

Train with Spark MLLib

This section covers how to train a language classifier using the text in the Tweets.

First, we need to featurize the Tweet text. MLLib has a HashingTF class that does that:

```
object Utils {
  ...

  val numFeatures = 1000
  val tf = new HashingTF(numFeatures)

  /**
   * Create feature vectors by turning each tweet into bigrams of
   * characters (an n-gram model) and then hashing those to a
   * length-1000 feature vector that we can pass to MLlib.
   * This is a common way to decrease the number of features in a
   * model while still getting excellent accuracy. (Otherwise every
   * pair of Unicode characters would potentially be a feature.)
   */
  def featurize(s: String): Vector = {
    tf.transform(s.sliding(2).toSeq)
  }

  ...
}
```

This is the code that actually grabs the tweet text from the tweetTable and featurizes it. K-Means is called to create the number of clusters and the algorithm is run for the specified number of iterations. Finally, the trained model is persisted so it can be loaded later.

```
val texts = sqlContext.sql("SELECT text from tweetTable").map(_.head.toString)
// Caches the vectors since it will be used many times by KMeans.
val vectors = texts.map(Utils.featurize).cache()
vectors.count()  // Calls an action to create the cache.
val model = KMeans.train(vectors, numClusters, numIterations)
sc.makeRDD(model.clusterCenters, numClusters).saveAsObjectFile(outputModelDir)
```

Last, here is some code to take a sample set of tweets and print them out by cluster, so that we can see what language clusters our model contains. Pick your favorite to use for part 3.

```
val some_tweets = texts.take(100)
for (i <- 0 until numClusters) {
  println(s"\nCLUSTER $i:")
  some_tweets.foreach { t =>
    if (model.predict(Utils.featurize(t)) == i) {
      println(t)
    }
  }
}
```

Run Examine and Train

To run this program, the following argument list is required:

1. YOUR_TWEET_INPUT - This is the file pattern for input tweets.
2. OUTPUT_MODEL_DIR - This is the directory to persist the model.
3. NUM_CLUSTERS - The number of clusters the algorithm should create.
4. NUM_ITERATIONS - The number of iterations the algorithm should run for.

Here is an example command to run ExamineAndTrain.scala:

```
%  ${YOUR_SPARK_HOME}/bin/spark-submit \
     --class "com.databricks.apps.twitter_classifier.ExamineAndTrain" \
     --master ${YOUR_SPARK_MASTER:-local[4]} \
     target/scala-2.10/spark-twitter-lang-classifier-assembly-1.0.jar \
     "${YOUR_TWEET_INPUT:-/tmp/tweets/tweets*/part-*}" \
     ${OUTPUT_MODEL_DIR:-/tmp/tweets/model} \
     ${NUM_CLUSTERS:-10} \
     ${NUM_ITERATIONS:-20}
```

Part 3: Apply the Model in Real Time

Spark Streaming is used to filter live tweets coming in, only accepting those that are classified as the specified cluster (type) of tweets. It takes the following arguments:

1. modelDirectory - This the directory where the model that was trained in part 2 was persisted.
2. clusterNumber - This is the cluster you want to select from part 2. Only tweets that match this language cluster will be printed out.

This program is very simple - this is the bulk of the code below. First, load up a Spark Streaming Context. Second, create a Twitter DStream and map it to grab the text. Third, load up the K-Means model that was trained in step 2. Finally, apply the model on the tweets, filtering out only those that match the specified cluster, and print the matching tweets.

```
println("Initializing Streaming Spark Context...")
val conf = new SparkConf().setAppName(this.getClass.getSimpleName)
val ssc = new StreamingContext(conf, Seconds(5))

println("Initializing Twitter stream...")
val tweets = TwitterUtils.createStream(ssc, Utils.getAuth)
val statuses = tweets.map(_.getText)

println("Initializing the KMeans model...")
val model = new KMeansModel(ssc.sparkContext.objectFile[Vector](
    modelFile.toString).collect())

val filteredTweets = statuses
  .filter(t => model.predict(Utils.featurize(t)) == clusterNumber)
filteredTweets.print()
```

Now, run Predict.scala:

```
 %  ${YOUR_SPARK_HOME}/bin/spark-submit \
     --class "com.databricks.apps.twitter_classifier.Predict" \
     --master ${YOUR_SPARK_MASTER:-local[4]} \
     target/scala-2.10/spark-twitter-lang-classifier-assembly-1.0.jar \
     ${YOUR_MODEL_DIR:-/tmp/tweets/model} \
     ${CLUSTER_TO_FILTER:-7} \
     --consumerKey ${YOUR_TWITTER_CONSUMER_KEY} \
     --consumerSecret ${YOUR_TWITTER_CONSUMER_SECRET} \
     --accessToken ${YOUR_TWITTER_ACCESS_TOKEN}  \
     --accessTokenSecret ${YOUR_TWITTER_ACCESS_SECRET}
```

Weather TimeSeries Data Application with Cassandra

This project demonstrates how to easily leverage and integrate Apache Spark, Spark Streaming, Apache Cassandra with the Spark Cassandra Connector and Apache Kafka in general, and more specifically for time series data. It also demonstrates how to do this in an asynchronous Akka event-driven environment. We use weather data and the existing hourly data format as the sample domain.

Time Series Data

The use of time series data for business analysis is not new. What is new is the ability to collect and analyze massive volumes of data in sequence at extremely high velocity to get the clearest picture to predict and forecast future market changes, user behavior, environmental conditions, resource consumption, health trends and much, much more.

Apache Cassandra is a NoSQL database platform particularly suited for these types of Big Data challenges. Cassandra's data model is an excellent fit for handling data in sequence regardless of data type or size. When writing data to Cassandra, data is sorted and written sequentially to disk. When retrieving data by row key and then by range, you get a fast and efficient access pattern due to minimal disk seeks – time series data is an excellent fit for this type of pattern. Apache Cassandra allows businesses to identify meaningful characteristics in their time series data as fast as possible to make clear decisions about expected future outcomes.

There are many flavors of time series data. Some can be windowed in the stream, others can not be windowed in the stream because queries are not by time slice but by specific year,month,day,hour. Spark Streaming lets you do both. In some cases, such as in the KafkaStreamingActor, using Spark with Cassandra (and the right data model) reduces the number of Spark transformations necessary on your data because Cassandra does the work for you in its cluster.

Overview

WeatherApp

This is the primary compute application that processes the raw hourly data from the Kafka stream. As soon as each hourly raw data is received on the particular Kafka topic, it is mapped to a custom case class. From there, two initial actions are done:

- The raw hourly data by weather station ID is stored in Cassandra for use by any other processes that want it in its raw state.
- The daily precipitation aggregate (0 to 23 hours of data) is computed and updated in a daily_aggregate_precipitation table in Cassandra. We can do precipitation in the stream because of the data model created and a Cassandra Counter used. Whereas aggregation of temperature require more input data so this is done by request with a station ID, year, month and day. * With things like daily precipitation and temperature aggregates pre-calculated and stored, any requests for annual aggregates, high-low or topK for example, can be easily computed via calls to specific Akka Actors handling each aggregation type (precip, temp, etc). These read the aggregated and persisted data from Cassandra and runs the computation through Spark. Data is returned to the requester in a Future - no threads are blocked. [Note, when in master with the correct URL of specific files available I can link them above so that a user can click on a specific block of code]

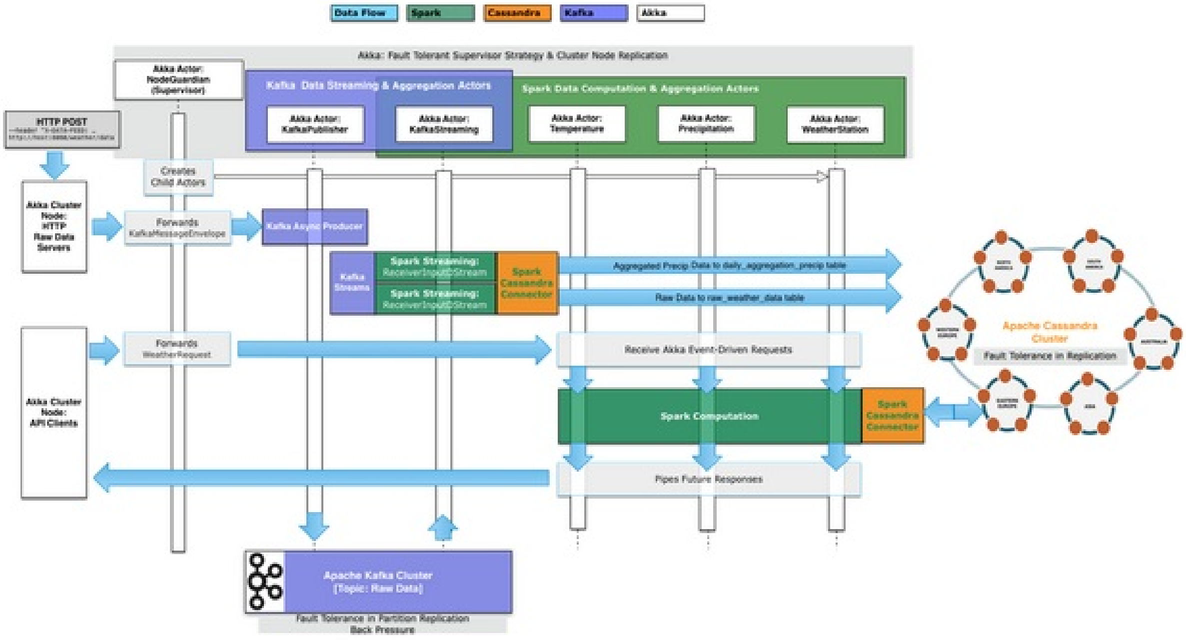
WeatherClient

- Represents a separately-deployed process that would typically be feeding data to Kafka and other applications or processes that would be sending data aggregation requests.
- Simulates real-time raw weather events received from a separate process which this sends to Kafka. It does this as a feed vs bursts on startup. The FileFeedActor in the client receives data per file, parses it to each raw data entry, and pipes the data through Akka Streams to publish to Kafka and handle life cycle.
- Exercises the weather app by sending requests for various aggregated data (topk, high-low, annual…) every n-seconds via the WeatherApiQueries Actor.

Architecture and Implementation Details

Asynchronous Fault Tolerant Data Pipeline

Raw hourly data for each weather station is ingested and published to a Kafka topic. This would be distributed on Kafka nodes in each Data Center where the Spark-Cassandra application are also deployed, as are the co-located Spark and Cassandra nodes.



Data is streamed from the Kafka nodes to the Spark Nodes for parallelized distributed data computation. The raw data is initially saved to a Cassandra keyspace and table. This stream from Kafka is then used for daily aggregation work in Spark, which is then persisted to several Cassandra daily aggregate tables. Now, future requests for data based on these daily aggregates (temperature, precipitation..) can now more quickly be computed. For example, requests for topK, high-low or annual precipitation for a specific weather station in a specific year (or year, month, day for high-low/topk) do not need to get the raw data but can start on the already aggregated data via Spark, now available on demand in Cassandra (replicated, fault tolerant and distributed across data centers).

Running the Application

There are many flavors of time series data. Some can be windowed in the stream, others can not be windowed in the stream because queries are not by time slice but by specific year,month,day,hour. Spark Streaming lets you do both. Cassandra in particular is excellent for time series data, working with raw data, transformations with Spark to aggregate data, and so forth. In some cases, using Spark with Cassandra (and the right data model) reduces the number of Spark transformations necessary on your data because Cassandra does that for you in its cluster.

When using Apache Spark & Apache Cassandra together, it is best practice to co-locate Spark and Cassandra nodes for data-locality and decreased network calls, resulting in overall reduced latency.

Setup

1. Download and install the latest Cassandra release
    - **Configuration Step:** Modify apache-cassandra-{latest.version}/conf/cassandra.yaml to increase batch_size_warn_threshold_in_kb to 64.
2. Start Cassandra.

    ```
    ./apache-cassandra-{latest.version}/bin/cassandra -f
    ```

    Note: If you get an error - you may need to prepend with sudo, or chown /var/lib/cassandra.

3. Run the setup cql scripts to create the schema and populate the weather stations table. Go to the timeseries data folder and start a cqlsh sell there:

    ```
    % cd /path/to/reference-apps/timeseries/scala/data
    % /path/to/apache-cassandra-{latest.version}/bin/cqlsh
    ```

    You should see:

    ```
    Connected to Test Cluster at 127.0.0.1:9042.
    [cqlsh {latest.version} | Cassandra {latest.version} | CQL spec {latest.version} | Native protocol {latest.versio
    Use HELP for help.
    cqlsh>
    ```

    Then run the script:

    cqlsh> source 'create-timeseries.cql'; cqlsh> quit;

See this Github repo to find out more about the weather stations table data.

Running the WeatherApp and WeatherClientApp

To Run from an IDE

1. Start com.databricks.apps.WeatherApp

2. Then start com.databricks.apps.WeatherClientApp

To Run from Command Line

1. Use SBT to run the app.

   ```
   % cd /path/to/reference-apps/timeseries/scala
   % sbt weather/run
   ```

   You should see:

   ```
   Multiple main classes detected, select one to run:
   [1] com.databricks.apps.WeatherApp
   [2] com.databricks.apps.WeatherClientApp
   ```

   Select option 1 to open the weather app.
2. Use SBT to run the client app. Run the same commands above, but select option 2.

About The Time Series Data Model

See this github repo to find out more about the Time Series Data Model